Robert Kovacsics
St. Catharine's College
rmk35

Diploma in Computer Science

Scheme (R^7RS) to Java bytecode compiler

May 10, 2016

Proforma

Name and College:	Robert Kovacsics, St. Catharine's College
Title of Project:	Scheme (R^7RS) to Java bytecode compiler
Examination and Year:	Diploma in Computer Science, 2016
Approximate Word-count:	8542
Project Originator:	Ibtehaj Nadeem's dissertation
Project Supervisor:	Dr John Fawcett

Original Aims

A compiler from the Scheme (R^7RS) programming language to Java bytecode supporting: basic language constructs; hygienic macros; Java interoperability; escape continuations and tail-call optimisation. Extensions, if time permits: runtime evaluation; full continuations and debugging features.

Work Completed

All of the futures highlighted above except debugging features.

Special Difficulties

Declaration of Originality

I Robert Kovacsics of St. Catharine's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed _____

Date May 10, 2016

Contents

List of Figures

List of Tables

List of Source Code Listings

Chapter 1

Introduction

1.1 Scheme Language and Runtime

This dissertation relies on a basic understanding of the Scheme programming language (R^7RS[1]), so a very brief introduction is given here. Scheme is a dynamically typed language with focus on simple semantics, and as such it has fewer but higher-level features than other languages. Of interest to us is that functions, continuations, and environments have "first-class" status meaning that they can be the result of expressions and stored in variables. First-class functions mean that Scheme supports the functional programming paradigm (but is not purely functional). First-class continuations allow Scheme to express control flow constructs such as coroutines and non-local exits (such as exceptions). First-class environments and homoiconic syntax—meaning that the abstract syntax of Scheme is representable within Scheme as a built-in datatype—are combined to help with runtime evaluation of code, the `eval` operator. The function evaluation is eager (function arguments evaluated before the function body) and has tail-call optimisation (also known as proper tail recursion) meaning that an unbounded number of tail-calls can be present.

The syntax of Scheme is made up of lists (all explicit apart from the top-level in a program) and literals. Thus, application is a list, and unlike in other languages, the operator is inside the list containing the arguments, this is known as the s-expression style, as shown in Listing 1.1. The semicolons start a comment until the end of the line. Syntactic constructs also follow this, which lends uniformity which is exploited by being

```
(if (= x 0)
    ; True case
    1
    ; False case
    (* x (fact (- x 1))) )
```

Listing 1.1: S-expression style.

able to define syntactic constructs (also known as macros). These macros are hygienic meaning that they neither introduce or capture bindings, this is best shown by the non-hygienic macro in Figure 1.2. The keywords in Scheme are not reserved, meaning that for example `if` could be overridden to be a different keyword or even to be a variable binding rather than a keyword.

```
int x = 1;                                                    // Output:
#define test_hygiene(M) {int y = 2; printf("x: %d\t", x); M;} // x: 3     y: 2
{ int x = 3;                                                  //
  int y = 4;                                                  // With hygiene:
  test_hygiene(printf("y: %d\n", y));                         // x: 1     y: 4
}                                                             //
```

Listing 1.2: Unhygienic macros in the C language.

1.2 Motivation

Different programming languages are suited to different tasks, but it can often be difficult to combine different programming languages together. Achieving this usually involves passing around machine-level data, which can lose type information and also depend on the underlying hardware, however using a virtual machine can ensure both portability and if type information persists in the bytecode format of compiled programs, also ease passing around typed data. The Java virtual machine (JVM) has type information in its bytecode format and furthermore also has runtime type information. The JVM is also a good target for languages as it is widely deployed on a large variety of platforms[2].

The native language for the JVM is Java, an imperative and object-oriented language with static types, so it would be complemented very well by a dynamically typed language that supports a different paradigm, such as the functional paradigm. For example, the idiomatic Java approach for walking a recursive data structure such as a parse tree, is the visitor pattern, as given in Listing 1.3, where angle brackets represent filenames. This has good software engineering properties as any change to the data structure—such as a new subclass—will be statically detected and force the programmer to extend the functionality of the program to handle the new subclass without missing any case, as could happen if one were to use the `instanceof` keyword with conditional tests.

However when prototyping software, safeguarding against an incorrect extension—as with the visitor pattern—becomes less important and the speed of development becomes much more important. This is because prototypes are disposable code built to learn, expose risk and offer a reduced cost for correcting mistakes[3]. As such, prototype code benefits from a more terse language, especially if it has dynamic interaction such as a read-evaluate-print loop (REPL), which provides a particularly fast turn around time. Scripting languages, such as Scheme, fill this gap, and Listing 1.4 gives the same example as idiomatic Scheme code.

As the unsuitability of Java for prototyping is due to the Java language that runs on the JVM, and not of the JVM, this means that we can implement a compiler from a scripting language such as Scheme to the JVM. Using the JVM as a common platform allows programmers to leverage existing code in either language and use the right language for the task at hand. It also allows the programmer to mix the programming languages in a way that gives more power than either separately, for example, Scheme does not provide threads or any other concurrent features but Java does, so having both on the JVM allows the programmer to fork multiple threads then from each, call the Scheme function that they want to execute, thus giving both concurrency and functional programming. Though it should be noted that as of Java version 8, Java also supports functional programming.

⟨visitor/Tree.java⟩ ≡

```
package visitor;
public interface Tree<T>
{ // Contravariant argument
  <U> U accept(Visitor<? super T, ? extends U> visitor);
}
```

⟨visitor/Visitor.java⟩ ≡

```
package visitor;
public interface Visitor<T, U>
{ U visit(Node<? extends T> node);
  U visit(Leaf<? extends T> leaf);
}
```

⟨visitor/Node.java⟩ ≡

```
package visitor;
public class Node<T> implements Tree<T>
{ Tree<T> left, right;
  public Node(Tree<T> left, Tree<T> right) { this.left = left; this.right = right; }
  public Tree<T> getLeft() { return left; }
  public Tree<T> getRight() { return right; }
  public <U> U accept(Visitor<? super T, ? extends U> v) { return v.visit(this); }
}
```

⟨visitor/Leaf.java⟩ ≡

```
package visitor;
public class Leaf<T> implements Tree<T>
{ T value;
  public Leaf(T value) { this.value = value; }
  public T getValue() { return value; }
  public <U> U accept(Visitor<? super T, ? extends U> v) { return v.visit(this); }
}
```

⟨visitor/LeafCountVisitor.java⟩ ≡

```
package visitor;
public class LeafCountVisitor implements Visitor<Object, Integer>
{ public Integer visit (Node<? extends Object> node)
  { return node.getLeft().accept(this) + node.getRight().accept(this);
  }
  public Integer visit (Leaf<? extends Object> leaf)
  { return 1;
  }
}
```

Listing 1.3: Simple tree walking example in Java.

```
; A node is a cons cell, and a leaf is anything else
(define (leaf-count tree)
  (cond ((pair? tree) 1)
        (else (+ (car tree) (cdr tree))) ) )
```

Listing 1.4: Simple tree walking example in Scheme.

1.3 Surrounding Area

Other implementations of Scheme for the JVM include SISC[4], which is an interpreter for the older R^{5}RS, and Kawa[5], which is a compiler for R^{7}RS, but does not allow an unbounded number of tail calls or support `call-with-current-continuation` fully. Other methods of getting Scheme on the JVM include using system calls such as with the Java Native Interface (JNI) to launch an existing Scheme implementation, but this limits portability and loses type information as this would be to the same effect as just passing around untyped data on the underlying hardware mentioned previously. There are also other virtual machines that could have been chosen such as the Common Language Runtime (CLR) from Microsoft or the Dis/Inferno Virtual Machine from Vita Nuova amongst many others. The choice of JVM was due to my familiarity with Java.

The implementation described here compiles R^{7}RS with both tail calls, and as an extension, has full support for `call-with-current-continuation`, though each of these come with overheads. In the evaluation chapter, the effect of having these overheads is measured and compared against the same implementation without supporting these features. For indication, the performance is also compared against a Java solution to the same problem, but performance was not a goal of this project. However, any solution used in this project needs to be practical so that we do not end up with something that cannot be used for real purposes.

Chapter 2

Preparation

2.1 Requirements for Deliverable Product

The core product that was agreed upon in the proposal (attached at the end) is a Scheme to Java bytecode compiler that supports an unbounded number of active tail calls, has partial support for `call-with-current-continuation`, can call Java and vice versa and also supports the hygienic macro pattern language. This requirement can be broken down into separate stages, this is usually done as shown in Figure 2.1 and provides a starting point for understanding the work that needs to be undertaken. The requirements for each

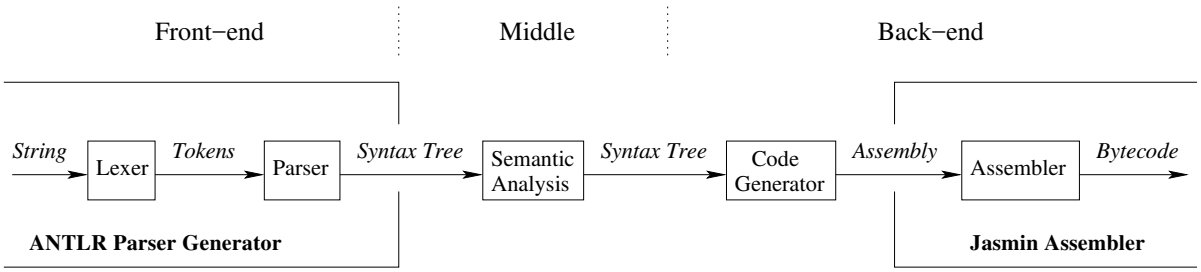

Figure 2.1: Stages of a compiler.

stage are detailed in the Scheme (R^7RS) specification[1], and part of my preparation was to look at the specification and understand it.

2.1.1 Front-end

The first two stages are given as context-free grammar, with the difference between the lexing and parsing that lexing outputs multiple data structures called tokens while parsing outputs a single data structure called the syntax tree. A distinction that exists in some other languages is that tokens can be described by regular expressions. This does not exist in Scheme, as Scheme has nested comments and commented expressions which need a tree structure to describe them. Nested comments are similar to block comments such as in C, but a nested comment can contain another nested comment, so that `/*/* invalid comment */*/` is not a valid as a block comment, but valid as a nested comment. This allows nested comments to comment code that already contains nested comments, but this still has problems, for example with commenting

`char* comment_end = "*/";` as inside the comment strings are not understood. For this reason Scheme has expression comments, which comment a syntactically correct Scheme expression, while being aware of the syntax inside the comment.

There is another problem with the nested comments of Scheme, as they interact with another piece of Scheme syntax that allows identifiers to contain spaces or any other character just like string literals. This produces a syntactic ambiguity as shown in Figures 2.2 and 2.3. Note that on the second derivation the comments are unattached to the identifier, as they are removed by before parsing.

⟨nested comment⟩ ⟶ `#|` ⟨comment text⟩ ⟨comment cont⟩* `|#`
⟨comment text⟩ ⟶ ⟨character sequence not containing `#|` or `|#`⟩
⟨comment cont⟩ ⟶ ⟨nested comment⟩ ⟨comment text⟩
⟨vertical line⟩ ⟶ `|`
⟨identifier⟩ ⟶ ⋯ | ⟨vertical line⟩ ⟨symbol element⟩* ⟨vertical line⟩ | ⋯
⟨symbol element⟩ ⟶ ⟨any character other than ⟨vertical line⟩ or \⟩ | ⋯

Figure 2.2: Excerpt from the Scheme syntax.

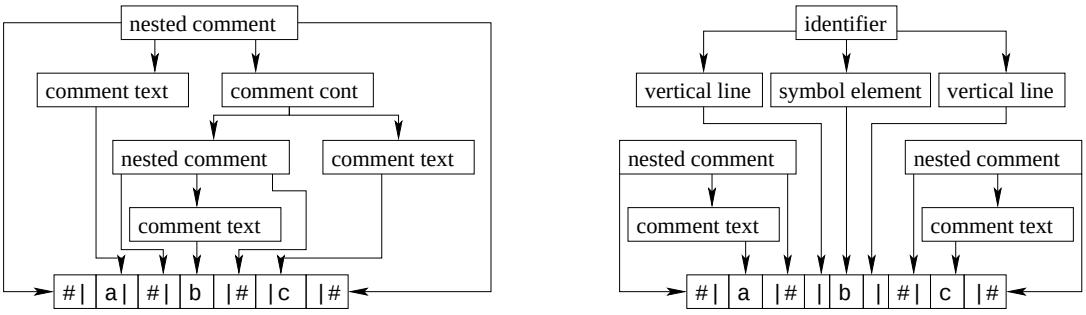

Figure 2.3: Ambiguity with nested comments and identifiers.

For the lexer and parser, I have chosen to use the ANTLR parser generator[6], as it was the most popular option in the Java community, and it is actively maintained. It is licensed under the BSD 3-clause licence. However during the project it had a small shortcoming; it did not support nested comments (but because of the ambiguity, neither does this compiler).

2.1.2 Middle

The output from the front-end is a syntax tree, for our compiler this is given by the ⟨datum⟩ grammar rule in the specification[1] and it consists of simple data such as booleans or numbers and compound data such as lists or vectors. This is the same as the `read` procedure returns and is unusual compared to other languages as it does not contain structures such as `if` or `lambda`, only the basic data-types of the language. This

is for two reasons, the first is that Scheme is homoiconic, meaning that the data uses the same representation as the code, and so it makes sense to first parse everything as data and reuse this code for the `read` operator. The second reason is that Scheme has no reserved keywords, so we cannot do further parsing until after the semantic analysis.

The other piece of semantic analysis that needs to be done for Scheme is the macro expansion, this as previously explained is hygienic, so that bindings are not introduced or captured at macro use time. The pattern language allows for repetition and alternation, which is shown in Listing 2.1. The alternation is given by the multiple rules for `and`,

```
(define-syntax and
  (syntax-rules ()
    ((and) #t)                    ; Conjunction of nothing
    ((and test) test)             ; Conjunction of one item
    ((and test1 test2 ...)        ; Conjunction of more than one item
     (if test1 (and test2 ...) #f) ) ) )
```

Listing 2.1: Recursive macro for the short-circuiting `and` syntax.

while the repetition is given by the use of the ellipses. It should be noted that ellipses can be nested, and match structurally, that is ellipses match to make a tree, not just a list of repetitions, this is shown in Listing 2.2.

```
(define-syntax structural-match
  (syntax-rules ()
    ((structural-match (x ...) ((y ...) ...))
     ((x y ...) ...) ) ) )

(structural-match (a b)
                  ((1 2)
                   (3 4) ) )
```

$\Longrightarrow$

```
; Expands into
((a 1 2)
 (b 3 4) )
```

Listing 2.2: Structural matching for macro patterns.

As the macro engine is closely tied to the identification of keywords, I have moved the implementation of the semantic analyser from work package 3 (parser) to work package 5 (macro engine). For this reason, I have implemented a small hygienic macro engine[7] during my research to manage the work for later. As the project was running one week late by the time I got to implement the macro engine, this allowed an efficient implementation that caught up with the intended schedule.

2.1.3 Back-end

The specification of how compiled code should behave is given in denotational semantics in the R^{7}RS specification. To understand what each language feature has to do, I have read it, however, I have not proved the compiler correct. The goal of the compiler is to generate JVM bytecode that has the same semantics as the expression that was compiled.

As an example, the semantics of the `lambda` expression is given in Figure 2.4. First, all denoted expressions take an environment (ρ), a dynamic point (ω, more on this later) and a continuation (κ), and return a command continuation, which is a function from

store (σ, memory) to an answer (the value of executing the computation in that store). Once the lambda is given the store, it tries to allocate a new bit of memory (line 2, *new* is functional, all invocations to $new\,\sigma$ give the same value), if it succeeds then control goes to line 3, if it fails then control goes to line 12 to report an error (the $p \to e_1, e_2$ is a notation for the conditional). The *send* on line 3 executes the continuation κ with the expressed value (X in `E` is the 'upcasting' of X to the expressed values E) that is a sequence (denoted by $\langle \cdots \rangle$). This sequence is composed of a new label ($new\,\sigma\,|\,\mathtt{L}$) 'downcast' to locations (which we know we can do due to the conditional) and a function. This function takes an arbitrary number of expressions (ϵ^*), a new dynamic point (ω') and a new continuation (κ') and first checks that we have the correct number of arguments on line 4, if not then we go to line 8 to report an error. Otherwise we allocate each argument to a location with the *tievals* that calls the outer lambda on line 5. This outer lambda has an inner lambda that is applied to the environment extended by binding the identifiers to the store locations (the memory model is identifiers bound in the environment to locations in store, which contain values). The body of the inner lambda is the semantics of Γ^* with the continuation of the semantics of E_0, in other words the lambda body can contain a sequence of Γ commands, but throws their value away (hence the use of the $\mathcal{C}$ semantic function and not the $\mathcal{E}$ semantic function) and returns the value that the last statement, E_0 returns. Line 11 is there to evaluate the command continuation returned by $send\,(\cdots)\,\kappa$, and it updates the store with an unspecified value—that is, up to the implementer—corresponding to the closure.

$\mathcal{E}[\![\texttt{(lambda (I*) } \Gamma^* \text{ } \mathrm{E}_0\texttt{)}]\!] =$

```
1   λρωκ . λσ .
2     new σ ∈ L →
3       send (⟨new σ | L,
4             λε*ω'κ' . #ε* = #I* →
5               tievals(λα* . (λρ' . C[[Γ*]]ρ'ω'(E[[E0]]ρ'ω'κ'))
6                             (extends ρ I* α*))
7                       ε*,
8               wrong "wrong number of arguments"⟩
9              in E)
10          κ
11          (update (new σ | L) unspecified σ),
12       wrong "out of memory" σ
```

Figure 2.4: Denotational semantics of `lambda`.

This semantics may look complex at first, but `lambda` is a complex example that is chosen to illustrate most concepts of the semantics. One concept that is not explained above is the use of the dynamic point. The dynamic point is a stack of before and after calls that must get called when the control flow changes such as by returning from a function. This can be compared to the Java `finally` keyword, but there is also a before version, as due to `call-with-current-continuation`, control can go back to a previous state. This can, for example, ensure a file is always opened inside a block and closed on exit. The dynamic point is illustrated in Figure 2.5 where full arrows denote the next element in the sequence and hollow arrows denote the value-of relationship. The dynamic point is used for the `call-with-current-continuation` and `dynamic-wind` operators.

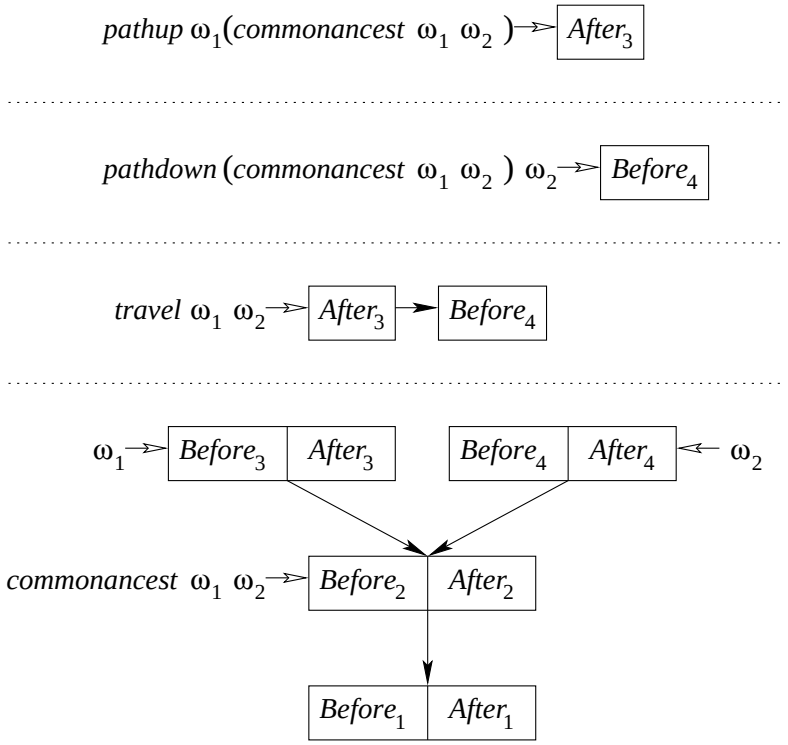

Figure 2.5: Illustration of dynamic point, and some functions on it.

$$
\begin{aligned}
&\mathit{list} : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C} \\
&\mathit{list} = \\
&\quad \lambda\epsilon^*\omega\kappa \,.\, \#\epsilon^* = 0 \to \mathit{send}\,\mathit{null}\,\kappa, \\
&\qquad\qquad \mathit{list}\,(\epsilon^* \dagger 1)\,(\mathit{single}(\lambda\epsilon \,.\, \mathit{cons}\langle\epsilon^* \downarrow 1, \epsilon\rangle\kappa))
\end{aligned}
$$

Figure 2.6: Type error in *list* auxiliary semantic function.

There is a type error in the semantics, as given in R^{7}RS[1], in the functions *tievalsrest* and *list* (a variadic function that returns a list of its arguments), the latter of which is shown in Figure 2.6 and my correction to it in Figure 2.7. The correction is in the recursive application of *list*, the dynamic point is left out, but since dynamic points should not be affected by constructing lists, I have found the correction of passing ω to recursive application sufficient. I have tried to contact the Scheme Working Group with regards to both the syntactic ambiguity and the semantic type error but I could not get through so I have posted my findings on the public Scheme list[8].

Apart from having to understand the Scheme semantics, I have had to research the

$$
\begin{aligned}
&\mathit{list} : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C} \\
&\mathit{list} = \\
&\quad \lambda\epsilon^*\omega\kappa \,.\, \#\epsilon^* = 0 \to \mathit{send}\,\mathit{null}\,\kappa, \\
&\qquad\qquad \mathit{list}\,(\epsilon^* \dagger 1)\,\omega\,(\mathit{single}(\lambda\epsilon \,.\, \mathit{cons}\langle\epsilon^* \downarrow 1, \epsilon\rangle\kappa))
\end{aligned}
$$

Figure 2.7: Corrected *list* auxiliary semantic function.

Java bytecode that runs on the JVM, the target of the compiler. The bytecode instructions are given in the Java Virtual Machine Specification[9]. However, I have decided to work with an assembly language rather than output bytecode directly, for ease of debugging. As the Java platform has no official assembler I have used the Jasmin assembler (BSD 3-clause license) which had all the features I needed, and was the oldest, so I expected it to be mature. Should this not be the case, it uses the de facto standard assembly format for Java so swapping it out could be done without too much trouble[10].

The JVM has two stacks: the value stack and the function stack. Almost all instructions operate on the value stack, and the function stack is only affected by the method invocation and return instructions. The JVM also has local variables, these are set to the function arguments—including the current object for non-static methods—and are also used by any local variables, thus making the JVM look more like a register machine. Just as with the Java language, all methods are present in classes and each class is in its own file, even more so as this includes inner classes. As an example of the assembly code, Listing 2.3 shows a simplified version of the identity function. The `java.util.function.Function` is the new Java 8 functional interface that requires us to define `apply`. This listing leaves out the creation of the `identityFunction` object that happens in a different file, and corresponds to the call to *new* in the semantics, as the function objects are used to hold closure values.

Scheme also requires an unbounded number of active tail-calls, but the JVM creates an activation record for each call. Two common approaches to get around this are converting tail-calls to loops and trampolining. Converting tail-calls to loops has the problem that in this implementation, functions are given their own files, and the JVM only supports transferring control between files via method invocations which add a new activation record to the call stack. Trampolining, on the other hand, uses suspended applications that are passed to the caller which resumes it, this way the stack is only increased by a constant amount. This is easily supported in Java by having suspended applications as classes.

2.2 Requirements for Development

To develop the software outlined above, some planning must be done beforehand to show that all the risks involved are managed so that the software and this dissertation can be successfully produced. Before the development stage of the project, there was a research phase which identified the techniques described in the previous section. As the deliverables of this, I have been producing documents detailing my knowledge of the necessary algorithms. This demonstrated my knowledge for my supervisor, highlighted potential problems early, served as documentation for later, and also acted as material for this dissertation. I was familiar with the programming language Scheme before I started the dissertation, as I have read the book Structure and Interpretation of Computer Programs (SICP) [11], and during part IB have taken Dr. Griffin's compiler development course.

During the supervisions, I have discussed the requirements for the product, and have decided to use the waterfall model of software development, with each work package outlined in the proposal as a separate module. This was because the requirements were known in advance, and the research meant that I have understood what needs to be done in each module, so there was no need for iterative development which approximates the

```
.class identityFunction
.super java/lang/Object
.implements java.util.function.Function

.field public x Lcompiler/RuntimeValue;

; Object constructor
.method public <init>()V
  aload_0
  invokenonvirtual java/lang/Object/<init>()V
  return
.end method

.method public apply(Lcompiler/RuntimeValue;)Lcompiler/RuntimeValue;
  ; For the Java bytecode verification
  .limit stack 3
  .limit locals 2

  ; Unpack arguments list, equivalent of tievals in the semantics ---;
  aload_1                                                          ;
  checkcast compiler/ConsValue                                     ;
  dup                                                              ;
  ; Call the 'public RuntimeValue getCar()' method on the cons cell ;
  invokevirtual compiler/ConsValue/getCar()Lcompiler/RuntimeValue; ;
  aload_0                                                          ;
  swap                                                             ;
  putfield identityFunction/x Lcompiler/RuntimeValue;              ;
  invokevirtual compiler/ConsValue/getCdr()Lcompiler/RuntimeValue; ;
  ; Check that we did not get to many arguments                    ;
  checkcast compiler/NullValue                                     ;
  pop                                                              ;
  ;----------------------------------------------------------------;

  ; Function body, get first argument and return it
  aload_0
  getfield identityFunction/x Lcompiler/RuntimeValue;
  areturn
.end method
```

Listing 2.3: Assembly code for the identity function `(lambda (x) x)`.

end product. For each module, I have chosen to adopt a test driven development to have an indicator of progress.

Against hardware loss, I have had other computers, the MCS at the computer laboratory or parent's computers at home. Against data loss, I have been making backups regularly and frequently (around after one hour's worth of work) to the MCS machines, or should they be unavailable, to a flash drive. Backups are also tested by extracting and compiling (compilation may fail if the backup represents a currently broken built code, but should only do so with the same message as on my personal computer).

During development, to allow reversal of unintended changes, I have been using the `git` version control system committing to an online repository each time a logical unit of change was done. This is different from backups as these are made to be able to restore earlier versions of the code, or to analyse the changes I have made such as when writing this document, whereas backups restore the current version.

The choice of the development language was Java, as this way I could use the compiler to be part of the runtime system, this is useful for when I came to implement the REPL and also made sense as Java has good software engineering properties such as a static type system, support from other tools such as the Maven build system[12] and has a large body of existing code such as the Jasmin assembler or the ANTLR parser generator that I have ended up using in the project. To reduce the boilerplate code in Java, I have used Project Lombok[13], which is a code generator controlled with Java annotations.

Chapter 3

Implementation

3.1 Getting Started

The preparation covered the requirements analysis of the waterfall model, so the next phase is implementation, and for each module I have written tests in advance, to help with the verification stage. Writing tests served as a guide as to what I needed to achieve per module and also gave data for evaluation, during development. However, the tests just show that for the given input the output is correct, so I tried to ensure that the inputs test corner cases, and when I have encountered a bug that the tests did not catch, I have added it to the test suite.

There are two kinds of tests I have used, unit tests which test individual functionality of components, such as name mangling or the data structure operations, and end-to-end tests which feed a program to be compiled to the compiler, and test the output of the compiler against known data. The end-to-end tests typically test one functionality at a time, though there are dependencies between end-to-end tests, for example testing simple function application on its own underlies all the other tests. Most of the end-to-end tests I have made up, however some are taken from the R^7RS specification. The unit tests are using the JUnit testing framework, which is a bit more tiresome to use as setting up internal data structures is harder than just writing Scheme code, but on the other hand it can be easier to provide it test data that looks at corner cases as it does not go through as many data transformations as end-to-end data does.

For the build system, initially I have used `make`, but this soon turned out to be insufficient in inferring dependencies between components of the program, so I have switched to Apache Maven, which is built for Java. It does a clean recompile every time but this is also the solution that I had to go for with `make` in the end. On the other hand, it manages the external dependencies of the project, which simplifies the build process on new machines, for example on the backup machine to test if the backup was successful.

To help me keep track of the things that I need to do, I have used a to-do file which I have set my editor to open with so that I do not lose track. I have also made use of the '`git` hooks' feature of the `git` version control system, which allows me to run arbitrary code—such as checking for comments that say this piece of code needs attention before the next commit. I have also tried to ensure that the whole structure of the project is comprehensible by having a glossary and a read-me file that detail the general ideas so that I know where to start if I come back to the project later.

Both extra software components that are not my own and form a part of the compiler (ANTLR and Jasmin) are under the 3-clause BSD license, so they can be combined with

the GPLv3[14] which I have decided to use in this project. The reason for using GPLv3 is because this product is not a library component but a tool, so there is more benefit in having extensions released under the same free-software license than there is in having it widely distributed with other software as a library.

3.2 Work Division

The modules of the project are detailed in the proposal, but a brief outline is shown in Table 3.1. However there have been some changes to the plan, such as the debugging extension was left out because I got ill for about a week around work package 9 and thought it would be better to be ahead with the dissertation than fall behind due to working on the debugger. Another change to the plan was prototyping the macro rewrite engine in work package 2, as I got fortunate and had plenty of time left after doing the planned research for then.

Another point of interest is that during the requirements analysis I have failed to factor in either of binding analysis or environments. The former was as I did not register the implications of having no reserved keywords and thus could not connect the front-end to the back-end, as the front-end did not have the required data. However, prototyping of the macro rewrite engine allowed me to do this in work package 5. The latter was due to other Lisps using the `eval` with an implicit environment—the current one—so first-class environments are not visible to the user. Since both of these were due to me approaching Scheme with assumptions from other languages, in the future I can avoid this by considering how the assumptions of other languages apply to the one I am implementing.

3.3 Overall Progress

The commit graph of Figure 3.1 shows the evolution of the code, the date labelled with the work package corresponding to the date interval on the x-axis against the lines of code changed on that date on the y-axis. In this and in further graphs the work package number falls in the middle of that work package. There are actually two lines, additions and deletions, and the coloured area shows the difference between them.

Another way of looking at this data is the 'chromatography' diagram (as it looks like a thin-layer chromatogram) of Figure 3.2. The x-axis is the same as before, but the y-axis is discrete, the presence of data on the y-axis means that the corresponding date

Front-end	Middle	Back-end
3. Parser	5. Macros	4. Basic code generation
	Extra: binding analysis	6. Java calls
	Extra: environments	7. Escape continuations
		8. Tail recursion
9. `read` operator		9. `eval` operator
		10. Full continuations

Table 3.1: Outline of modules, numbers reference work packages.

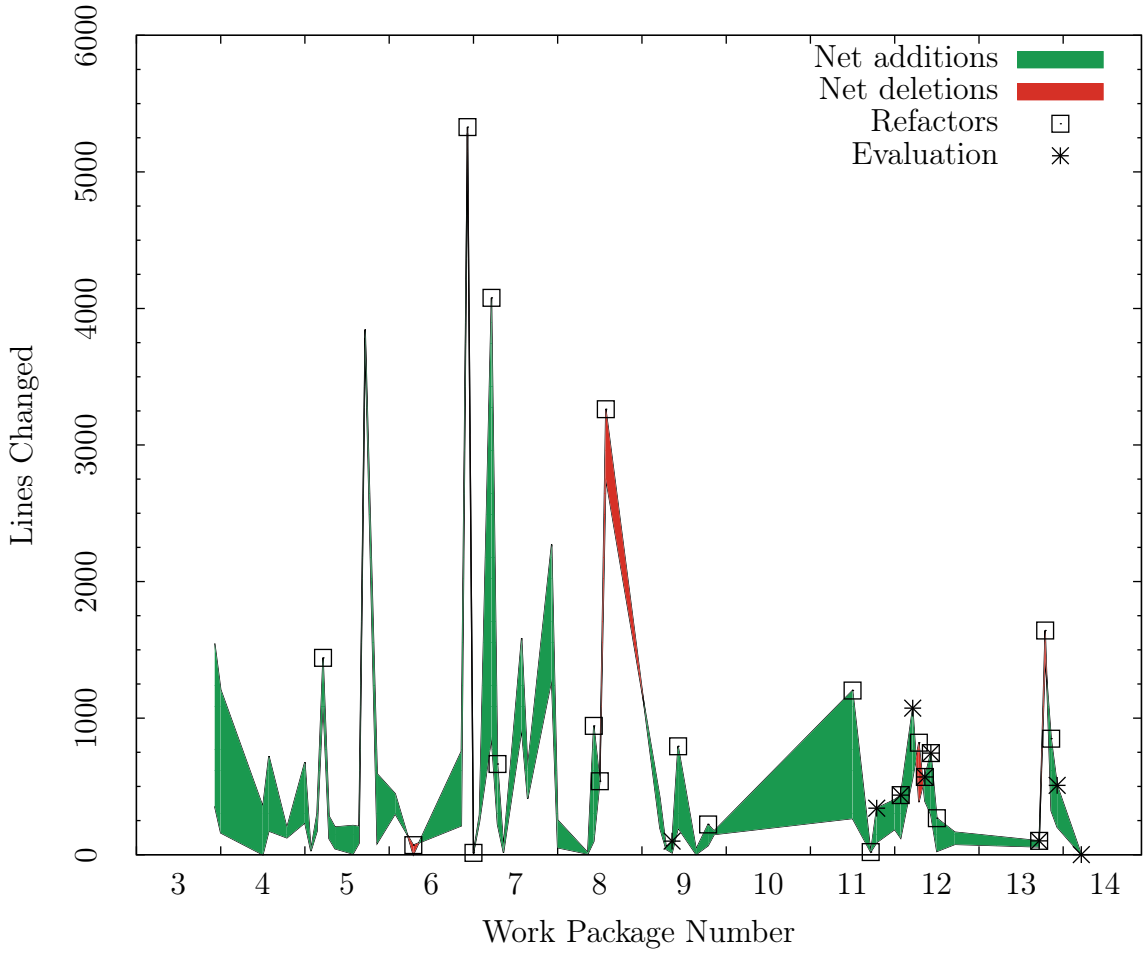

Figure 3.1: Commit graph of changes by line.

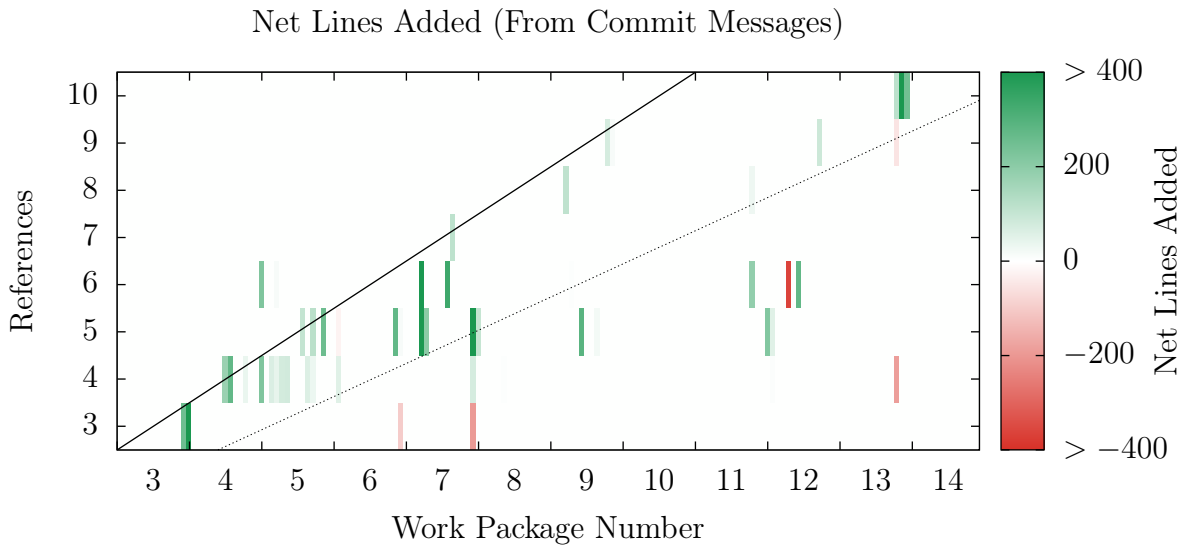

Figure 3.2: 'Chromatography' of work done.

contained a commit with the net changed lines given by the colour intensity. It does not include refactors, dissertation work or evaluation, as they don't belong to any specific work package.

Considering the commit graph, the initial green area of the graph is large because of the licence file, and other such additions rather than extreme productivity at the start. The large spikes that have little coloured area enclosed by them correspond to refactoring or moving code, for example the largest spike is when I have changed from using `make` files to using Maven. All commits that are refactors are tagged with a square. These refactors did not come for free, but using automation they were not prohibitive and they improve the code and allow future changes to be made more easily. The best example of this is the refactor in work package 8 that has net deletions rather than net additions; this change was uniting the front-end syntax tree with the back-end/runtime data structures, as this removed duplication.

The commit graph is only an approximation of the overall effort, but shows that during the project I have tried to maintain a steady amount of work done rather than cram work. It also shows changes to the repository that are due to the evaluation or verification stage of the waterfall model, either in the form of adding more test cases or correcting newly discovered bugs.

For the second diagram, the solid line goes through the middle of the work package on either axis to indicate the optimal case, while the dashed line indicates a damped least-squares fit. No correlation coefficient is included as the line is for illustration, a high correlation would just indicate that I have not gone back to earlier work packages, which I have had to do because of later discovered bugs and in some cases generalisation of the architecture. One example of this is that during the research phase I have made a mistake in my approach to closures by copying the value into the closure, and not the reference. This was the cause for the change in work package 4 around the time when I was supposed to be working on work package 13. The reason for this is me not following the semantics enough when I was doing the research for using Jasmin in the first work package.

The purpose of the regression line is to show that, if we factor in later maintenance to code, my estimation was off by a rate of about 20 days against one week as suggested in the proposal. However, this does not include the extra week I gained due to good fortune, during which I have spent prototyping the macro engine, but this would not noticeably increase that estimate as the total implementation time was 22 weeks. This is useful information that I would take into account were I to redo the project, or for estimating the complexity of future projects.

One of the reasons I have still been able to deliver the project on time is because I have allocated slack to catch up, multiple so that it could help me with the slant in the regression, rather than just a constant offset. Also, I have moved the dissertation forward, across work package 10 and 11 so that should the dissertation also need more time, I could still hand it in on time. Then later on when it became apparent that I could hand it in on time, I have done work package 10.

Further points of interest on the 'chromatography' graph are the trailing commits of work package 5 (on the y-axis) which were bug fixes and other work on the macro pattern matching engine until the start of work package 12 (on the x-axis) when I have realised that my approach was doing the 'wrong thing' and I have implemented the 'right thing', details of which are explained later. Strictly speaking this went beyond the requirement for the compiler, but it made more sense to do the right thing rather than patch up what

I currently had and risk further patches in the long run.

Around the same time, I was also working on using message passing rather than just exposing the java reflection interface to the user, this is the last three commits for work package 6 (on the y-axis). This also went beyond the requirement for the compiler, but it made the Java calls more pleasant and helps me avoid implementing Scheme libraries (which was also not part of the proposal) as I could just use the Java libraries.

3.4 Technical Detail

3.4.1 Syntax Transformations

Once the project got started, there was a lot of technical detail to get right, such as inferring the syntactic structure. (Recall that due to the lack of reserved keywords parsing only takes us to lists or vectors of identifiers and literals.) Fortunately, Scheme is very regular in that all of the syntax is in prefix form, so actually all of the syntax can be matched by using the macro matching engine. The macro engine is, on the other hand, a much more complex part of the compiler, which can be broken down into two parts: the pattern matcher and the hygienic rewriting engine.

The pattern matching engine used is based on a simple backtracking approach, however unlike other backtracking regular expression engines[15], this one does not have the exponential (in the size of the pattern) worst case. The reason for this is the restriction on the way patterns can be expressed, they have to have alternation on the outside and can only have one repetition per list/vector, so that the matches are unambiguous. Some regular expressions cannot be expressed this way, e.g. `(a|b)`* because we cannot push the repetition inside the alternation as it leaves out for example `ababa`. This means that the only backtracking done is by one repetition per list (and matching multiple lists adds to the cost, not multiples it) which is $O(N^2)$ for a list of size N. As an example consider matching `(a ...)` against the input string `(a`^{n}`)` (superscripts denote repetition for the input). The matcher first tries to match `()` at a cost of 1 and fails, then tries `(a)` at a cost of 2, `(a a)` at a cost of 3 and so on until it matches n lots of '`a`'s at a cost of n, with an overall cost of $1 + 2 + \cdots + n = n(n + 1)/2$. Then there is a simple optimisation that we apply, which is when we fail to match `(a a)`, and try `(a a a)`, we remember that we could match the first two '`a`'s (otherwise this pattern could never match) and resume matching the last '`a`', giving an overall cost of $O(N)$. On top of this, alternation also just adds to the cost, and for any pattern there is only a fixed, finite amount of alternation so the overall worst case is still $O(N)$ for an input of length N.

A different way of looking at this is that the regular expression `(a?)`^{n} which is expressible in n characters normally, is expressible only in 2^n characters using Scheme's pattern matcher as you have to enumerate the possibility of each `a` being matched or not matched. In this case, a human would remove all the n ways of matching a single `a` and so on, but consider `a?b?c?`$\cdots$ which has no such redundancy. The optimisation described in [15] would enable the compiler to discard alternations much faster, but on the other hand would not improve the performance as reading in the pattern (for example the 2^n characters of `(a?)`^{n} with alternations pushed outside) dominates the complexity.

The hygienic rewriting engine is based off the one described in [7] which is an improvement on the original Kohlbecker's algorithm. The improvement is in combining the rewriting and expansion to lower the running time from $O(N^2)$ to $O(N)$ (the comparison is between the time taken by the two approaches, not with respect to the input as macros

can be recursive and not terminate in the worst case). The algorithm works by rewriting identifiers in the template that do not appear in the matched pattern and have a different binding at use and definition time by fresh identifiers and binding these identifiers to the meaning in the definition environment. Identifiers that do appear in the pattern are replaced with what they matched in the input. This however does not consider multiple matches as is possible with repetition.

Adding repetition to this involves the use of perfect binary trees, which have the distance between the root and any leaf the same for all leaves. This condition means that all matches of an identifier have the same ellipsis level. So now substitutions captured by pattern matching map identifiers to trees (or leaves if it is not inside an ellipsis, i.e. if we are not matching a repetition), and during rewriting we replace identifiers for trees. Then when we encounter a subtemplate that is followed by an ellipsis, we fold the last nodes to a leaf in the tree returned by that subtemplate. This use of binary trees is the right approach, earlier I have tried to use flat lists for repetition, which was harder to work with and left me with bugs.

As an example, look at Figure 3.3, which illustrates Listing 2.2. The first two frames illustrate what the substitution is for x and y, then the third frame shows one fold, because we have encountered an ellipsis. In the fourth frame, the substitution—which is, and has to have, the same shape as the previous frame—the match of x is merged with the previous frame, and in the last frame we get another fold for the final result. Building such substitutions when we match patterns is not shown, but is easily achieved by adding new leaves to the perfect tree (while keeping the invariant of same depth, it is an error if we cannot do so).

3.4.2 Environments and Libraries

The Scheme memory model consists of an environment mapping from identifiers to store locations and a store mapping from locations to values. The environment is a first-class value in Scheme, its use is with the `eval` function. This environment can be extended by loading in bindings from other libraries similarly to Java imports, except these can happen at runtime too. These are implemented using the Java class loader facility to find the environment, then using reflection to instantiate it and then load values from it.

Similarly, the related `eval` function responsible for runtime evaluation of expressions also first compiles the expression to Java bytecode, then use the class loader to load in the bytecode and again use reflection to instantiate and run the bytecode. I have mentioned that I don't expect the compiler to be distributed with other software, but to use the `eval` feature the compiler will have to be distributed along with the software using it, as it invokes the compiler during runtime. However this is not a problem for most software, as this feature was also not present in earlier Scheme (R^4RS and older) so a lot of other Scheme programs don't use this feature.

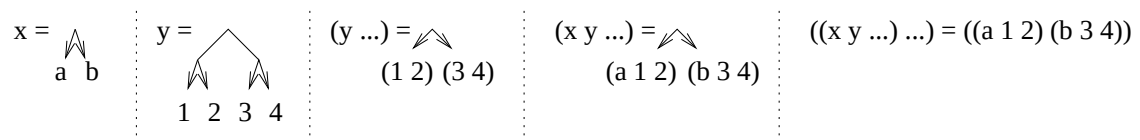

Figure 3.3: Ellipses in the template.

3.4.3 Code Generation

The architecture of the code generator is outlined in Figure 3.4. The OutputClass and subclasses represent output files, the main class is the entry point into the compiled program, the inner classes represent Scheme functions (and closures) and their output implements the Java 8 `java.util.function.Function` interface containing the `apply` method. The method represents a bytecode method (such as class, object constructor or the `apply` method), and contains a list of strings, the converted instructions. The statements are simplified Scheme expressions while the instructions represent Java bytecode. The reason for statements using methods rather than methods containing statements is that statements are an intermediate form whereas methods are bytecode methods.

When the statements are adding instructions to the methods, the instructions also simulate the number of items on the value stack, and when a method is converted to string, it verifies that the number of items on the value stack is consistent with the return value. This is for ease of debugging, as spotting a missing or erroneous instruction can be very difficult.

Using trampolining is then implemented on top of statements by returning a value encapsulating an application with specific arguments (a CallValue) for the application statement. There is a trampoline on the call stack so that when it gets such a value, it evaluates the call until it get a non-CallValue, and returns that. When we have continuation points, we push a new trampoline on to the stack by calling Trampoline.bounce, so that any non tail-calls are evaluated without forgetting where to continue from once the call returns.

Full continuations are then implemented very similarly to trampolines—for application it returns a call object—the change is in continuation, as instead of calling a new trampoline, we add a new stack frame. This is summarised in Table 3.2.

	No tail recursion	Trampolining	Full continuations
Call	Native Java call	Return CallValue	Return CallValue
Continuation	Nothing	Call Trampoline.bounce	Step program counter, push new stack frame

Table 3.2: Summary of tail-call approaches.

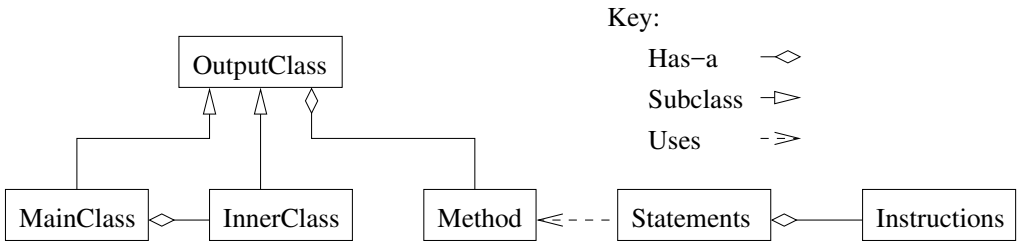

Figure 3.4: Architecture of code generation.

Chapter 4

Evaluation

4.1 Evaluation Reproducibility

All of the evaluation was done on a machine running the software given in Table 4.1. NixOS is a Linux distribution that has reproducible, declarative configurations[16], so installing the version given in the table should give the same software that I have used. For reference, the NixOS configuration I have used for testing is also included in the appendix on page 31, which I have used to generate an isolated virtual machine—with the machine using virtual time—to decrease the effect of interrupts on timing analyses, and also boosted the priority of the test process so that it does not get interrupted so often. For graphs with error-bars, the error bars represent the unbiased variance $\overline{S}_n/(n-1)$ in the data around the given mean $\overline{X}_n$ of n samples. For these I have used the online formula below, to decrease numerical errors.

$$\overline{X}_n = \overline{X}_{n-1} + \frac{X_n - \overline{X}_{n-1}}{n} \qquad \text{and} \qquad \bar{S_n} = \overline{S}_{n-1} + \left(X_n - \overline{X}_{n-1}\right)\left(X_n - \overline{X}_n\right)$$

4.2 Working Features

Table 4.2 gives a brief overview of the simple automated tests I have used to check the correctness of the compiler features. Some of these tests are from the formal specification, but some I have made up. When I have made the tests, I followed the formal specification carefully to avoid bugs being present in the tests, and when I have found bugs in the program, I have added a test to cover the bug, with the intent that the bugs in the tests and the program will not end up overlapping.

Most of the tests are straight forward, but some require a bit of explanation.The tail-call tests use the looping combinator to tail-call ad infinitum, and the test passes if we

Software	Version
NixOS Linux	16.09.git.32b7b00 (Flounder)
OpenJDK	1.8.0_76-00
Apache Maven	3.3.9
GNU bash (as /bin/sh)	4.3.42(1)

Table 4.1: Software versions for evaluation.

Feature	Work package	Reference
Quote	3	tests/quote
Application	4	tests/application
Closures	4	tests/closures
Conditionals	4	tests/conditional
Store mutation	4	tests/store
Macro hygiene	5	tests/macros
Macro defining macro	5	tests/macro-defining-macro
Scheme and Java calls	6	tests/java-calls
Exceptions	7	tests/exceptions
Tail calls	8 and 10	tests/tail-calls
Runtime evaluation	9	tests/runtime-eval
Imports		tests/imports

Table 4.2: Simple functionality tests.

have not ran out of stack after one minute. A much more satisfying—but manual—test is running the test manually (`cd tail-calls; ../run test.scm`) and then using the quit signal (control-\) which prints the stack, to verify that the stack is not growing. Testing imports does not fit into any work package, but the default Scheme environment is empty, so all other tests need to import bindings and thus rely on it.

A much more interesting test of `call-with-current-continuation` feature relies on a somewhat subtle interaction with local variables to print the sequence `@*@**@***`$\cdots$, with the number of `*` symbols increasing forever. This is shown in Figure 4.1 which is originally from [17]. The idea is that the continuation captured by yang contains the value held in yin, and at each call to yin we set the new yin to be the old yang, so we get a continually increasing chain of continuations that print `*` symbols. The benefit of this, apart from intellectual satisfaction, is that there are many ways to get this wrong (such as having the call `yin yang` overwrite all of the yin values, in which case we get `@*@*****`$\cdots$, without ever getting another `@`) so it catches out many potential bugs.

```
(define (get/cc) (call-with-current-continuation (lambda (c) c)))
(let* ((yin
        ((lambda (cc)
          (display "@") cc)
         (get/cc) ) )
       (yang
        ((lambda (cc)
          (display "*") cc)
         (get/cc) ) ) )
  (yin yang) )
```

Listing 4.1: The yin-yang puzzle.

Using all of the features described above, starting up a read-eval-print-loop is a simple matter of running code such as in Listing 4.2. The following is a transcript from a REPL session, where the greater-than sign at the start of the line indicates user input. The tran-

script also shows that even without mutation, having `call-with-current-continuation` means that we do not have referential transparency, as in this case we have mutated the value stored in `cc`.

```
> (define cc (call-with-current-continuation (lambda (x) x)))
#<unspecified>
> cc
#<continuation: ...>
> (cc 1)
#<unspecified>
> cc
1
```

```
(import (scheme base) (scheme read)
        (scheme eval)(scheme write)
        (scheme java))

(define interaction-environment (mutable-environment '(scheme base)))

(let loop ()
  (displayln "> ")
  (writeln (eval (read) interaction-environment))
  (loop))
```

Listing 4.2: Simple code for a REPL.

4.3 Performance of Features

Earlier, I have said that the performance of the macro engine is linear with respect to the size of the input, so Figure 4.1 shows the time taken for `(a ...)` to match $(\texttt{a}^n)$ against n. The error bars show the variance, and the line is a linear regression on the points, with the Pearson product moment-correlation coefficient given as r. The graph shows 100 data points, each data point sampled 500 times. The decrease in the variance after the fifth data point is due to the Java virtual machine optimising the code after a set number of method invocations (this is by default $1,500$ method invocations, see `CompileThreshold` in the `java(1)` manual page). Also for times longer than about 5 milliseconds we see an increase in variance due to the scheduler interrupting the JVM.

Figure 4.2 shows the difference in execution time between Java and Scheme (using our own call stack for Scheme calls) when calculating the Fibonacci series using the naive algorithm which has exponential complexity (base $\varphi = (1 + \sqrt{5})/2$). The exponential functions have been fitted using the damped least-squares fit and give us an estimate for the slow-down factor of $1.9 \times 10^{-3}/4.5 \times 10^{-6} \approx 420$, which is quite a lot. However this is with us maintaining our own stack. Figure 4.3 shows the execution time for the same program using no tail-call optimisation and trampolining, with a slow-down factor of about 71 and 110 respectively.

This raises the question of why there is such a dramatic slow-down, Figure 4.4 shows the answer. The axes show: the time taken to access local variables; the time taken to

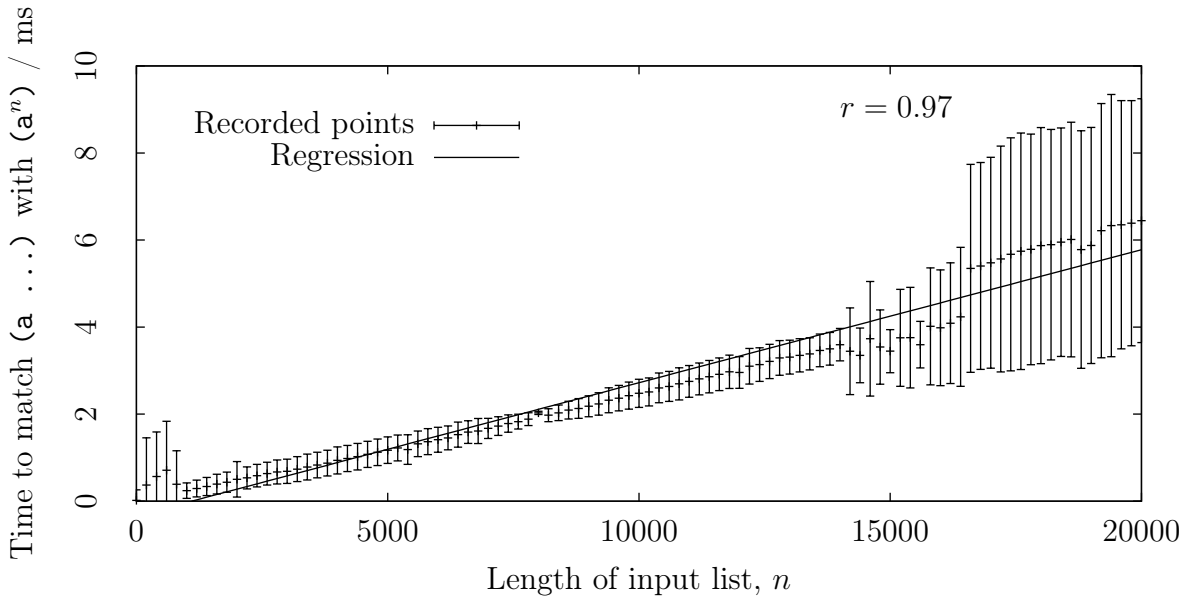

Figure 4.1: Matching pattern `(a ...)` with input string $(\texttt{a}^n)$ against n.

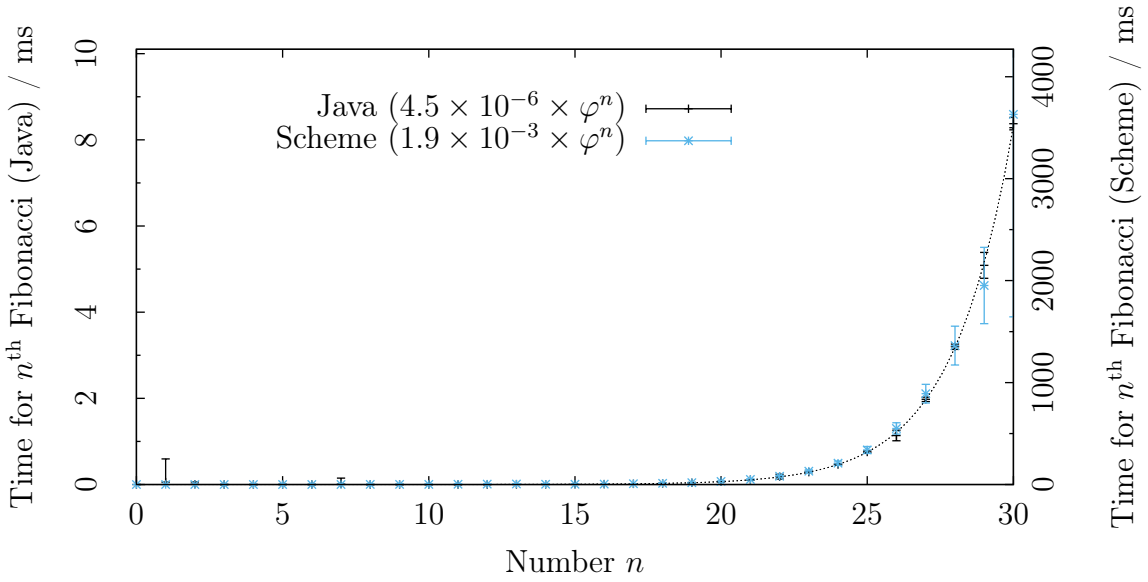

Figure 4.2: Naive Fibonacci in Java and Scheme (using Scheme call stack).

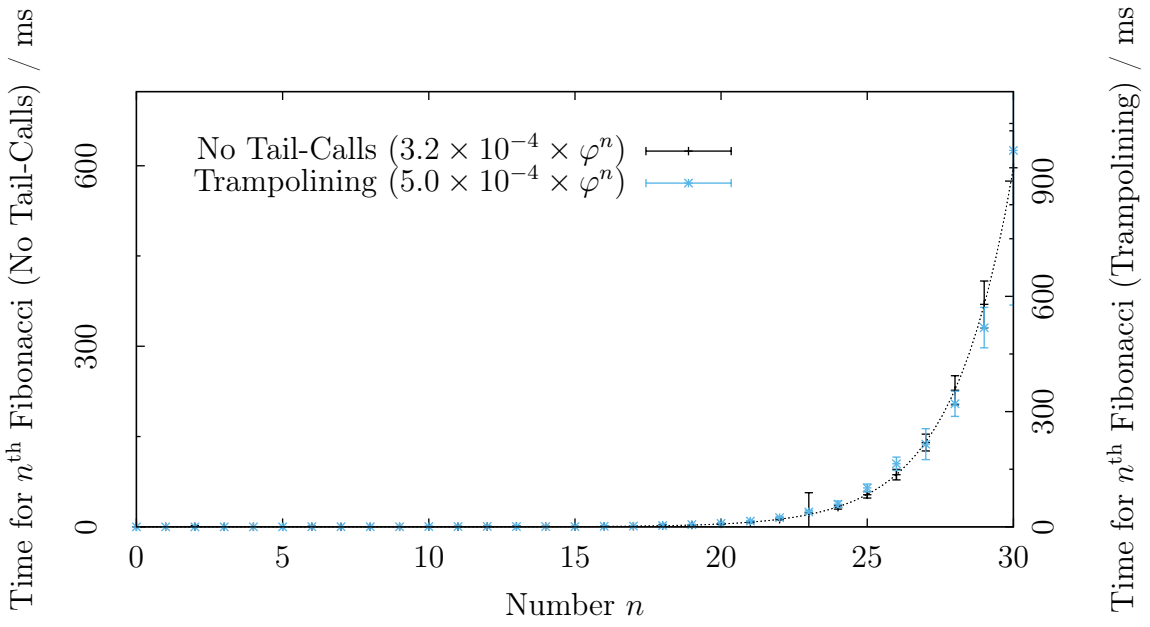

Figure 4.3: Naive Fibonacci in Scheme using no tail-calls and trampolining.

create objects such as numbers, or calls (as we are using our own stack, to indicate that we have a call we return a call object); initialisation time; time spent in library routines; time spent on deciding whether the value returned by the predicate of a branch is true or false; and time spent type-checking arguments, as Scheme uses dynamic types. It shows that the largest bottle-neck is the time taken to create objects, this is something that Java does not encounter as in this case Java uses primitives. However we cannot do this as due to the dynamic types, we need a common supertype to return, and primitives do not fit into the type hierarchy. The second largest bottle-neck is that we use field variables for method variables, as opposed to the JVM's local variables. The reason for this is that we need method variables to be preserved when we are using our own stack and as such re-entering methods, but the JVM does not do this. Field variables are slower as we need to load the object of which the field belongs to—this is a double dereference due to the JVM garbage collector's indirection—and only then can we load the value we want. The library call overhead in part consists of all the other overheads shown in the graph, as due to the abstraction they present, their code has not been profiled separately. Also as stated before, efficiency was not the main goal of the project.

Another metric that we can take is the overhead of calling Java from Scheme and vice versa. In this case I have included each data point on Figure 4.5 rather than using mean and variance, as the overhead was too small compared to the variance because of interrupts, and interrupts also occurred regularly so repetition did not reduce the variance. It shows that calling Java from Scheme is the slowest, this is because the compiler uses late binding, implemented with reflection, then calling Scheme from either language has similar overhead as Scheme is the dominating factor, and finally calling Java from Java has the least overhead. The split in calling Java from Java is actually due to the previously mentioned runtime optimisation, it is more apparent on the raw data as the two are divided in time.

The final timing analysis is the time spent on each of the compiler stages outlined before, as a proportion of the total time of the compilation, given in Figure 4.6. This data is gathered from the compilation of the meta-circular interpreter from Structure and

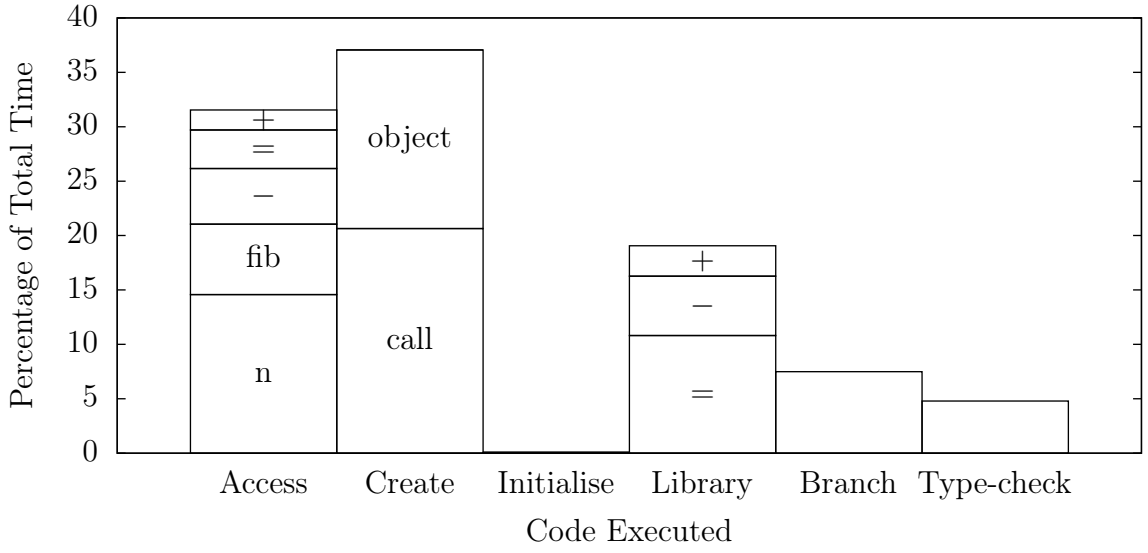

Figure 4.4: Profile of calculating the 30th Fibonacci number.

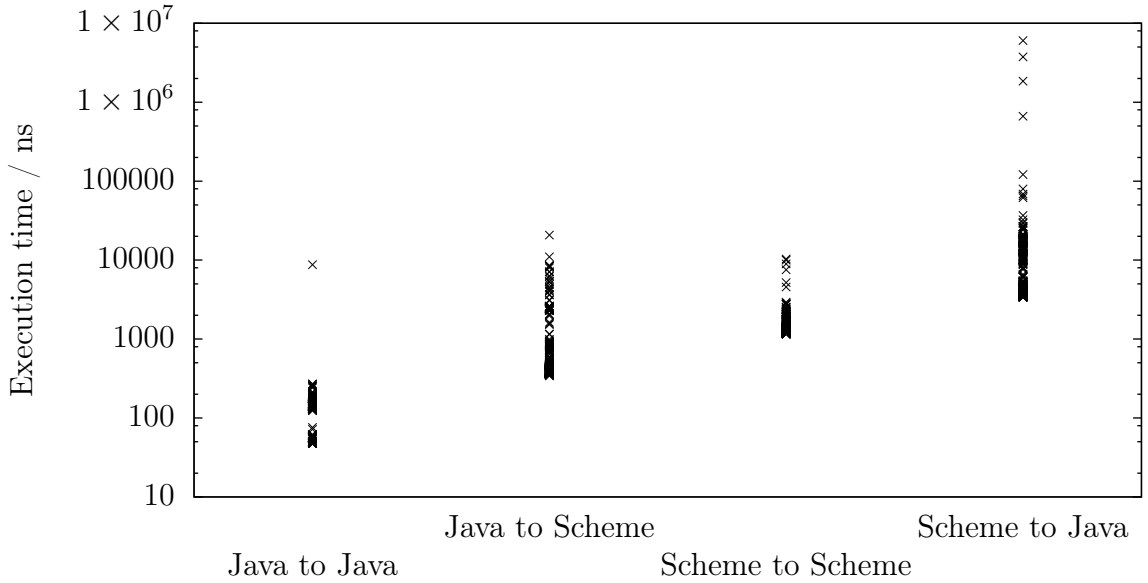

Figure 4.5: Java and Scheme call overhead (logarithmic y-axis).

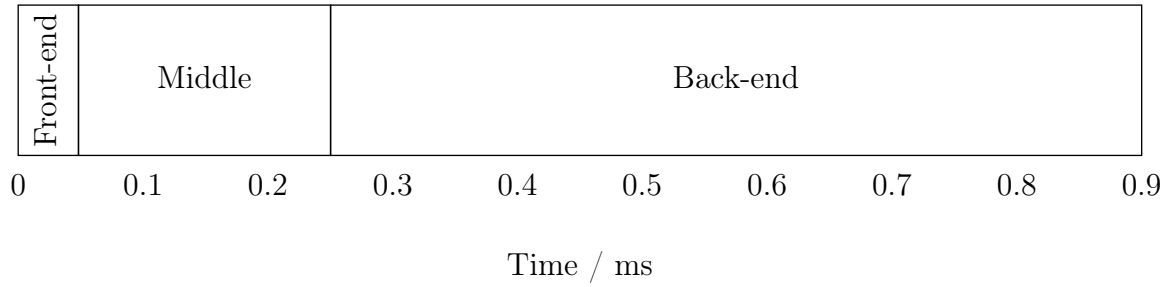

Figure 4.6: Proportion of time spent on each compiler stage.

Interpretation of Computer Programs (SICP)[11], which is a program of about 370 lines. It is in fact a Scheme interpreter for the subset of Scheme used in SICP, and being able to compile and run it is important as it shows a practical example of Turing completeness as the subset is itself Turing complete. The front-end took the least time, and the code executed here was reading in the file and parsing it, which uses code generated by ANTLR. Most of the back-end code was also executing code that I did not write, the Jasmin assembler which does its own parsing as it is reading in the assembly code in text form.

Chapter 5

Conclusion

Apart from the optional debugging extension, the project had successfully implemented all the other features from the proposal. The reason for not implementing the debugging extension is due to lack of time, as despite the slack I have given myself, I was slowly getting behind the schedule. This was for mostly due to having to go back to parts of the project and fix bugs, in the most severe cases this was due to earlier misunderstandings that meant I had to redesign parts of the compiler to do the right thing. A considerable amount of the development time was spent refactoring, this was due to earlier mistakes in the program design. While this enabled later code to be written more easily, it also put the current work behind schedule. However the solution to this for future projects is not necessarily doing more design, and starting later, as in this case I felt that at the start of implementation I have spent too much time putting off writing code, but perhaps a better solution is to quickly prototype parts of the project so that I would end up exposing problems earlier.

As an example, initially I have started using Java lists for the Scheme lists as well, so that I would be able to use their already implemented features to my benefit, but later it turned out that they map poorly on to Scheme lists. Because I have realised this late there was already a lot of code that depended on Java lists and the refactor was large, though I was able to automate some of it.

The project also has several shortcomings, one of which is shown in the evaluation chapter, the slowness of the compiled code. Part of this could be optimised, but it is also in part due to Scheme constructs not mapping on to the JVM as efficiently as Java constructs do. The most obvious example of this is the inability to access and modify the call stack of the JVM. This is because Java does not have features such as co-routines, but possibly also because of security considerations. A possible Part II project could then involve making a virtual machine for functional languages, or a virtual machine for non-linear control flow such as given by first-class continuations, possibly based on labels as described for the theoretical programming language GEDANKEN[18] (this is what the compiler does but doing this at the virtual machine level may bring about speed improvements).

Another important feature missing from the compiler is understandable error messages. At the moment, it just prints the Java stack trace when something (for example a type error) goes wrong, the problem with this is that it shows a lot of irrelevant details about the compiler which makes extracting the actual error more difficult, and also for example the user may not know that a ClassCastException means a type error. Similarly, a macro stepper would enable the user to verify that the macros that the user wrote work

as intended.

In terms of usability, the standard library that comes with the project is probably too small for real world projects, though being able to use Java calls means that the Java standard libraries could be used. On the other hand this would not make for a very exciting project.

While not part of the Scheme specification, some Scheme implementations support more powerful macros with a primitive known as `syntax-case`[19]. This would extend macros to do more than just tree transformations, for example predicated pattern matching or compile-time code evaluation.

Also not part of the specification is concurrency, but this could fit in well with existing features, such as first-class continuations to avoid having to write code in the continuation-passing style, which for example in JavaScript causes a problem known as call-back hell. One project looking at this is 8sync[20], which uses GNU Guile, another Scheme implementation.

Or possibly Scheme could be extended to have more object-oriented features, for example similar to CLOS with possibly a meta-object protocol on top, as described in The Art of Metaobject Protocol[21].

Bibliography

[1] Steven Ganz et al. *Revised^7 Report on the Algorithmic Language Scheme*. 6th July 2013. URL: `http://www.scheme-reports.org/2015/working-group-1.html` (visited on 3rd Oct. 2015).

[2] Oracle Corporation. *Learn About Java Technology*. URL: `http://www.java.com/en/about/` (visited on 23rd Mar. 2016).

[3] Andrew Hunt and David Thomas. *The Pragmatic Programmer*. 1999, pages 53–56. ISBN: 978-0-201-61622-4.

[4] Scott G. Miller. *SISC – Second Interpreter of Scheme Code*. URL: `http://sisc-scheme.org/` (visited on 28th Feb. 2016).

[5] Per Bothner et al. *Kawa: The Kawa Scheme language*. URL: `https://www.gnu.org/software/kawa/` (visited on 28th Feb. 2016).

[6] Terence Parr and Sam Harwell. *ANTLR (ANother Tool for Language Recognition)*. URL: `http://www.antlr.org/` (visited on 15th Oct. 2015).

[7] William D. Clinger and Jonathan Rees. "Macros that work". In: *Conf. Rec. 18 ACM Symposium on Principles of Programming Languages*. 1991.

[8] Robert Kovacsics. *comp.lang.scheme group. Questions about R7RS Syntax and Semantics*. 22nd Jan. 2016. URL: `https://groups.google.com/forum/#!topic/comp.lang.scheme/kYV0pO35Oe4`.

[9] Tim Lindholm, Frank Yellin, Gilad Bracha and Alex Buckley. *The Javaő Virtual Machine Specification. Java SE 8 Edition*. 13th Feb. 2015. URL: `http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html` (visited on 25th Mar. 2016).

[10] Jonathan Meyer, Daniel Reynaud and Iouri Kharon. *Jasmin*. URL: `http://jasmin.sourceforge.net/` (visited on 15th Oct. 2015).

[11] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. 2nd edition. MIT Press, 1996, page 186. ISBN: 978-0-262-51087-5. URL: `https://mitpress.mit.edu/sicp/` (visited on 28th Mar. 2016).

[12] The Apache Software Foundation. *Maven – Welcome to Apache Maven*. URL: `https://maven.apache.org/` (visited on 27th Apr. 2016).

[13] Project Lombok Authors. *Project Lombok*. URL: `https://projectlombok.org/` (visited on 23rd Mar. 2016).

[14] Free Software Foundation. *Various Licenses and Comments about Them*. URL: `https://www.gnu.org/licenses/license-list#GPLCompatibleLicenses` (visited on 3rd Apr. 2016).

[15] Russ Cox. *Regular Expression Matching Can Be Simple And Fast*. URL: https://swtch.com/~rsc/regexp/regexp1.html (visited on 4th Apr. 2016).

[16] Eelco Dolstra et al. *About NixOS*. URL: http://nixos.org/nixos/about.html (visited on 28th Apr. 2016).

[17] David Madore. *call/cc mind-boggler*. URL: https://groups.google.com/forum/#!msg/comp.lang.scheme/Fysq_Wplxsw/awxEZ_uxW20J (visited on 29th Apr. 2016).

[18] John C. Reynolds. "GEDANKEN—A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept". In: *Commun. ACM* 13.5 (May 1970), pages 308–319. ISSN: 0001-0782. DOI: 10.1145/362349.362364. URL: http://doi.acm.org/10.1145/362349.362364.

[19] R. Kent Dybvig. *Writing Hygienic Macros in Scheme with Syntax-Case*. Technical report. Indiana Computer Science Department, 1992.

[20] Christopher Allan Webber et al. *8sync: Asynchronous Programming Made Awesome*. URL: http://www.gnu.org/software/8sync/ (visited on 28th Apr. 2016).

[21] G. Kiczales, J.D. Rivières and D.G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991. ISBN: 978-0-262-61074-2.

Appendix

⟨configuration.nix⟩ ≡

```
# Build using
# NIXOS_CONFIG="`pwd`/configuration.nix"
# NIX_PATH="nixpkgs=/path/to/nixpkgs/repository/at/revision/32b7b00:$NIX_PATH"
# export NIXOS_CONFIG NIX_PATH
# nixos-rebuild build-vm

# Run with exported shared directory (ideally project root)
# Note, using VM clock, not real clock
# log-in with name: "guest" and password: "vm"
# SHARED_DIR=/home/kr2/Dokumentumok/PartII-Project/part-II-project/
# export SHARED_DIR
# ./result/bin/run-vmhost-vm -rtc 'base=utc,clock=vm'

# Then /tmp/shared is mounted to wherever you have set $SHARED_DIR

{ config, pkgs, ... }:
{ environment.systemPackages = with pkgs; [ jdk maven ];

  fileSystems."/".label = "vmdisk";

  networking.hostName = "vmhost";
  users.extraUsers.guest = {
    isNormalUser = true;
    uid = 1000;
    password = "vm";
    group = "wheel";
  };
  users.extraUsers.root = {
    password = "root";
  };
  security.sudo = {
    enable = true;
    wheelNeedsPassword = false;
  };
  system.stateVersion = "16.09";
}
```

Listing 5.1: Configuration of NixOS virtual machine.

Proposal

Robert Kovacsics
Part II
St. Catharine's College
rmk35

Diploma in Computer Science Project Proposal

Scheme (R^7RS) to Java bytecode compiler

May 10, 2016

Project Originator: Ibtehaj Nadeem's dissertation

Resources Required: See attached Project Resource Form

Project Supervisor: Dr John Fawcett

Director of Studies: Dr Sergei Taraskin

Overseers: Dr Markus Kuhn and Prof Peter Sewell

Introduction and Description of the Work

This project involves implementing a compiler from a subset of Scheme (R^7RS) small language specification [1] to Java bytecode in such a way that Java can interoperate with Scheme. The Scheme language is a multi-paradigm programming language supporting functional and meta programming (in the compile-time transformation sense). In contrast Java supports object-oriented programming and reflection (which can also be seen as a form of metaprogramming).

The subset of Scheme for the core project contains the base library with escape continuations, but without supporting full continuations with infinite extent. (That is continuations cannot get outside of the `lambda` passed to `call/cc`.) The Scheme specification mandates the compiler to allow unlimited tail calls, which will pose a challenge on the JVM. The core for the Java interoperation will support calling Scheme code as static methods of some class, and Scheme code will be able to perform static, virtual and non-virtual method calls.

The limited support for continuations is due to the Java virtual machine not allowing the inspection and modification of the call stack, so an extension will be looking at implementing full continuations. Another extension could be enabling the use of reflection on Scheme inside Java and reflection on Java inside Scheme. As an extension, making a debugger, possibly via the REPL as in other Scheme implementations, has also been suggested by Dr Kuhn.

Resources Required

I plan to use my own computer, it has an i7-2670QM 2.2GHz CPU, 8GB RAM, 1TB Disk and Linux OS. I will use git/GitHub for backing up and version control to protect me from data loss. Apart from having a feature branch, I will also have a topic branch to allow me to make commits such as “end of day” or “about to clean up, proceeding with caution” whose purpose is not to be commits that will necessarily be merged onto the feature branch, but to minimise the amount of work I will need to recover should I have software/hardware failure (or just having removed the wrong file).

However I shall also make daily copies to the MCS machines (this will also leverage that MCS machines are also backed up on their own, so should be resilient). I also recognise that back-ups are only useful if they are operational, so I will be using `rsync` with checksums to transfer backups and recompile the latest working source on the MCS machines. (Git also uses checksums in commits to ensure the integrity of the commit data.)

In case of software failure due to upgrades, my distribution provides roll-back but in any case I am using programs already installed on MCS machines. In case of hardware failure I will use the MCS machines. Should this happen I will be making back-ups to pen drives (and onto Google docs, as I have lost pen drives before) so that not all of the back-ups are in one place. I accept full responsibility for my machine and I have made contingency plans to protect myself against hardware and/or software failure.

In terms of software libraries, I will be using ANTLR [2] and Jasmin [3], both of which are freely available and also run using Java so should not need any installation onto the MCS machines in case my machine becomes unusable.

Starting Point

I have taken Dr Tim Griffin's course on Compiler Construction and over the summer read Structure and Interpretation of Computer Programs which teaches Scheme and also has a small section on interpreting and compiling Scheme.

Substance and Structure of the Project

I propose to write the core of the compiler in Java, but after the core of the compiler is done, also write some of it (such as some of the library component) in Scheme. The core part of the project will consist of implementing the following Scheme (R^7RS) subset:

- Hygienic macros

  These extend the language at compile-time to include more special forms. Hygiene means that variable capture is avoided when macros are substituted and also each free identifier in a macro refers to the binding at the time when the macro was specified.

- Proper tail recursion

  Tail calls can be recognized and then converted to branches at the bytecode level.

- The base library

  This should not take too long as it is a small base library, but should provide enough functions to run some of the existing benchmarks out there.

- Escape continuations and exceptions

  Escape continuations are continuations that are used similarly to throw/catch exception handling. Formally, they never exit the `lambda` into which they were passed as arguments, such as:

  ```
  (call/cc (lambda (continuation)
             (continuation 'error)))
  ```

  and thus also don't allow re-entry into a piece of code as calling the continuation will give control to where the continuation is no longer in scope.

- Calling Java from Scheme and vice versa

  This can, on a bytecode level, be achieved by one of the `invoke` JVM instructions. On a language level Scheme methods will need to be mapped into Java namespace, this can be achieved by putting them as static methods in the class files. This will need name-mangling as Scheme allows identifiers to be composed of a much wider set of characters than Java. Calling Java from Scheme is a little more complex as it can be a static, virtual or non-virtual call, but each of these could be done by separate methods taking strings as Java namespace and method names to call if necessary.

Implementing a compiler can be broken down into lexical, syntactic and semantic analysis and code generation. For this project, the lexical and syntactic analysis are not considered interesting for the scope of the project and as such will be performed by another third party utility such as ANTLR [2]. Thus the bulk of the work will include semantic analysis and code generation, along with fully understanding the denotational semantics of Scheme for the implementation (though proof of correctness is not planned). Also, code generation is expected to generate ASCII description of the bytecode, as this will give easier to debug outputs. The conversion from the ASCII to binary Java class files can be handled by Jasmin [3]. This conversion is not hard, but having a tool doing it gives me more time for harder problems.

Then on top of the core project, I propose the following extensions

- Full support for continuations

  For this to work, we in some way will need to capture the call stack. As the Java virtual machine does not let us manipulate the call stack, we need to use our own call stack instead, which will involve modifying the code that calls other Scheme methods.

- Reflection for support of `eval`

  For this the compiler can be bundled as a runtime environment and called on strings (`read` then `eval`) to produce Java classes. These classes can then be loaded in and their main method called via reflection. This should allow a simple read-eval-print loop (REPL) to be made as a demonstration of its functionality.

- Simple debugging

  This is an extension suggested by Dr Kuhn. It would tie in the above extension by using the REPL to provide the interaction. The features of the debugging would be a `trace` method to record function entry, nesting level and arguments; a `break` method that when called, allows the REPL to print values and step in execution. There are two ways to implement Java debuggers, the first one is using the Java API, which would need C++ and allow me to interact with the JVM easily (for example to walk the call stack). The second one would work at a higher level and instead of using the JVM breakpoint instructions, it would use a method invocation into the REPL. While this is less efficient and also does not give the ability to walk the call stack, I propose to use it as this will allow me to easily re-use the previous extension and be of a smaller scope. I feel that choosing the C++ option would rather be a larger effort as it consists of making a new project, which would be a large risk for an extension late in the project.

Success Criteria

The following should be achieved:

- Running benchmarks to demonstrate language features (not performance)

  This aims to demonstrate that the language features have been implemented and are working well. Efficiency is not a main aim of this project, so the actual performance comparisons are not important. For comparison, other Scheme implementations

can be used. A suite of benchmarks can be found at The Larceny Project [4], though some of those benchmarks require more of the standard libraries than this implementation will provide.

- Correct tail call optimization

  A tail recursive program which without tail calls would run out of stack should be able to run without problems in this Scheme implementation. A profiler could also be attached to the JVM to visually show this.

- [Extension] Full support for continuations. This should be able to run programs such as the following puzzle:

```
(define (get/cc) (call/cc (lambda (c) c)))
(let* ((yin
         ((lambda (cc) (display #\@) cc)
          (get/cc)))
       (yang
         ((lambda (cc) (display #\*) cc)
          (get/cc))))
    (yin yang))
```

  which prints the sequence `@*@**@***`$\cdots$ with increasing number of asterisks. This should also give me the opportunity to measure the overhead of maintaining our own stack instead of using the JVM stack for method calls.

- [Extension] A read-eval-print loop to show the compilation and reflection in action.

  This should give me the opportunity to show off the compiler for the supervisor's report. This is deemed to be successful when it can support any of the features that the compiler supports.

- [Extension] Tracing and stepping through execution.

  This extension will also give me the opportunity for showcasing the compiler. This is successful when I can step through a program and inspect the value stack. It is not expected to support walking the call stack.

Timetable and Milestones

1. 26th October–8th November — Michaelmas term

Familiarise myself with Java bytecode and the R^{7}RS report. Write notes on the primitive expression types (those defined as denotational semantics and not macros: `lambda` abstraction, application, `if` conditional and mutation with `set!`) and also generate Java bytecode of small examples of those expressions to have an idea of what code generation will have to output.

Milestone: Made and sent notes and example codes in to supervisor.

2. 9th November–22nd November — Michaelmas term

Read about hygienic macros [5] and produce a document that can be sent to the supervisor, detailing how macros with hygiene can be implemented efficiently. This work may not take long time if no extra reading is necessary, in which case implementation can start early.

Milestone: Made and sent notes about hygienic macros to supervisor.

3. 23rd November–6th December — Michaelmas term

Michaelmas term ends 4th December.

The front-end of the Scheme compiler: parsing and semantic analysis. As a work package this is light, but this is because I expect there to be some overhead with getting the project started. Also, some design decisions here may affect the extensibility of the compiler for later on, so it is worth taking time to implement this cleanly.

Milestone: Front-end of the compiler (parsing and semantic analysis) implemented.

4. 7th December–20th December — Christmas vacation

Implement code generation supporting the primitives (`lambda` abstraction, application, `if` conditional and mutation with `set!`). These are primitives in terms of which other constructs can be (in terms of semantics) defined. (Practically, the calling of Java methods will also be important step later on, as it will help implement data structures and I/O.)

Milestone: A compiler supporting the above mentioned primitives.

5. 21st December–3rd January — Christmas vacation

Implement the hygienic macro system. This will require a transformation engine to avoid variable capture and preserve the bindings inside the macro to be those at the time of the macro declaration. I have not done such work before so I have placed this early in the schedule in case it presents difficulties.

Milestone: Defined the extra syntactic constructs that appear in the R^7RS specification [1].

6. 4th January–17th January — Christmas vacation

Lent term starts 12th January.

Add the ability to call Java methods from Scheme and vice versa. This is expected to be light work, and was chosen to act as slack for the major milestone. This package will also feature implementation of the base library functions that were missed out before.

Milestone: Defined base library data-types (integer, boolean, character, string, vector, list) in terms of their Java equivalent.

Major milestone: Core project, lacking continuations and tail recursion, implemented.

7. 18th January–31st January — Lent term

Progress report deadline on 29th January.

Prepare progress report and presentation. Implement escape continuations. These will be implemented as exception throwing/catching and these then can be used to implement Scheme's exception mechanism as macros.

Milestone: Progress report submitted before the deadline. Ability to use escape continuations and raise/guard exceptions.

8. 1st February–14th February — Lent term

Progress report presentation on 4th, 5th, 8th and 9th February

Implement tail recursion optimization. This requires modifying code generation to output a different code for tail calls that ensures the size of the stack does not grow. This is expected to be light work and act as slack for the next major milestone.

Milestone: Presentation prepared. A tail recursive program that runs out of stack space without tail recursion optimization should run fine with the optimization.

Major milestone: Completed core of the work package.

9. 15th February–28th February — Lent term

Write the introduction and preparation chapter of the dissertation. Implement the `eval` construct. This is expected to use the existing compiler as mentioned before, to compile the code into a class file, then load it in via reflection. A method avoiding writing the code to a file may be investigated as usually code `eval`uated is small and files can be a big overhead.

Milestone: Draft of introduction and evaluation chapter submitted to my supervisor. Ability to `read` and `eval` previously existing code as strings.

10. 29th February–13th March — Lent term

Lent term ends 11th March.

Write the implementation chapter of the dissertation. Implement support for full continuations. This means implementing my own stack for the compiler. This will also need to handle `dynamic-wind` more carefully as now we can re-enter a piece of code we have exited so we need to make sure we run the before block (for the after block a finally construct should suffice).

Milestone: Draft of implementation chapter submitted to my supervisor. Ability to run the yin-yang puzzle in the success criteria.

11. 14th March–27th March — Easter vacation

Slack to catch up if required. Otherwise polish the dissertation using the feedback and implement the debugging extension. This is expected to support breakpoints and stepping through the code. I expect revision to slow down progress from here onwards.

Milestone: Ability to step through the evaluation of an expression.

Major milestone: All code finished.

12. 28th March–10th April — Easter vacation

Evaluate the finished code and write the draft of the evaluation and conclusions chapter. Continue polishing the rest of the dissertation.

Milestone: Draft of evaluation chapter sent to my supervisor.

13. 11th April–24th April — Easter vacation

Easter term starts 19th April.

Polish the dissertation into final form.

Milestone: Final form of the dissertation submitted in to my supervisor for overview.

14. 25th April–8th May — Easter term

Incorporate any feedback into the dissertation and submit it. As this is the last work package, I expect little work by here but if needed it can act as slack to allow the dissertation to be submitted before the deadline, even in case of delays.

Milestone: Submitted the dissertation before the dissertation deadline on 13th May.

Bibliography

[1] Steven Ganz et al. *Revised^7 Report on the Algorithmic Language Scheme*. 6th July 2013. URL: `http://www.scheme-reports.org/2015/working-group-1.html` (visited on 3rd Oct. 2015).

[2] Terence Parr and Sam Harwell. *ANTLR (ANother Tool for Language Recognition)*. URL: `http://www.antlr.org/` (visited on 15th Oct. 2015).

[3] Jonathan Meyer, Daniel Reynaud and Iouri Kharon. *Jasmin*. URL: `http://jasmin.sourceforge.net/` (visited on 15th Oct. 2015).

[4] William D. Clinger. *The Larceny Project, Benchmarks*. URL: `http://www.larcenists.org/benchmarks2015.html` (visited on 15th Oct. 2015).

[5] William D. Clinger and Jonathan Rees. "Macros that work". In: *Conf. Rec. 18 ACM Symposium on Principles of Programming Languages*. 1991.